DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

SOFTWARE RELIABILITY STUDIES

By

Mary Ann Hoppa, Research Associate

and

Larry W. Wilson, Principal Investigator

Final Report
For the period ended October 31, 1994

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

Under
Research Grant NAG-1-750
Kelly J. Hayhurst, Technical Monitor
ISD-System Validation Methods Branch

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, VA 23508

November 1994

Software Reliability Report

NAG 1-750
Larry Wilson
Department of Computer Science
Old Dominion University
Norfolk, VA 23529-0162

There are many software reliability models which try to predict future performance of software based on data generated by the debugging process. Our research has shown that by improving the quality of the data one can greatly improve the predictions. We are working on methodologies which control some of the randomness inherent in the standard data generation processes in order to improve the accuracy of predictions.

Mary Ann Hoppa, a PhD student at ODU, has used a LIC version with ten known bugs to build a database consisting of empirical reliabilities for each of the 1024 nodes of the partial debugging graph [1]. These reliabilities were each found by subjecting a debugging variant of the software to a million input cases. This data base has been used to analyze the effect of the order in which bugs are removed on the reliability predictions of four well known models [2]. This work found that the models were indeed sensitive to the order in which the bugs are found. We are in the process of preparing a second paper which will analyze the effects of using a surrogate oracle in the data collection. We are interested in the surrogate since our previous data collections required a gold version and thus were useful only as laboratory techniques. The surrogate oracle is expected to give good approximations and to be useful in a productive software engineering environment. If the approximations prove to be highly accurate we will have significantly improved the accuracy of predictions by existing software reliability models with only a minor increase in debugging cost.

Pam Bowman, a MS student at ODU, is developing a part of a partial debugging graph using a different LIC specimen. This is being done to analyze programmer dependency relative to the work done by Hoppa and will also investigate alternative phenomena. Also, Weimin Shi, a MS student at ODU, is working on simulation studies for the Goel- Okomoto model. These studies will parallel those done earlier by Wenhui Shen for the Jelinski-Moranda and Geometric models [3]. These simulation studies complement the empirical studies in that both are investigating the benefits of replicated data from the debugging process as opposed to the single sample data currently used by the models.

Future work will incorporate the information previously generated into an integrated package with the new results. It is also our hope to further investigate some of these ideas using the GCS environment and versions, which are part of the on going NASA-LARC experiment.

References

1. Wilson, Larry W. and Shen, Wenhui, "Software Reliability Perspectives", Old Dominion University Computer Science Department # TR-87-035, 1987.

2. Hoppa, Mary Ann and Wilson, Larry W., "Some Effects of Fault Recovery Order on Software Reliability Modelsi", To appear in the Proceedings of ISSRE 94. Also ODU CS Technical Report #TR-94-28.

3. Shen, Wenhui and Wilson, Larry W., "Simulation Studies of Software Reliability Models ", NASA Contractor Report 181889. Also released as ODU CS TR-89-10

DEPARTMENT
OF
COMPUTER SCIENCE

Technical Report # TR–94–28

Some Effects of Faulty Recovery Order On Software Reliabiltiy Models

Larry Wilson and Mary Ann Hoppa

Old Dominion University
Computer Science Department
Norfolk, VA 23529

October 15, 1994

Old Dominion University
Norfolk, VA 23529-0162

Technical Report # TR–94–28

Some Effects of Faulty Recovery Order On Software Reliabiltiy Models

Larry Wilson and Mary Ann Hoppa

Old Dominion University
Computer Science Department
Norfolk, VA 23529

October 15, 1994

Some Effects of Fault Recovery Order on Software Reliability Models

Mary Ann Hoppa
5425 Lawson Hall Key
Virginia Beach, VA 23455
804-464-4121

Larry W. Wilson*
Department of Computer Science
Old Dominion University
Norfolk, VA 23529
804-683-3084

Abstract

Ultrareliable software is required for life critical applications and the assessment of that reliability requires ever greater accuracy from reliability models. Feedback from controlled, repeatable experiments is particularly needed to assess and extend the efficacy of existing software reliability models. Also, traditional approaches allow the experimenter to formulate predictions using data from one realization of the debugging process. Thus for ultrareliable applications it is necessary to understand the influence of the fault recovery order on the predictive performance of reliability models.

Our contribution is twofold in that we describe an experimental methodology using a data structure called the debugging graph and apply this methodology to assess the robustness of existing models. The debugging graph is used to analyze the effects of various fault recovery orders on the predictive accuracy of several well-known software reliability algorithms. We found that, along a particular debugging path in the graph, the predictive performance of different models can vary greatly. Similarly, just because a model "fits" a given path's data well does not guarantee that the model would perform well on a different path.

Further we observed bug interactions and noted their potential effects on the predictive process. We saw that, not only do different faults fail at different rates, but that those rates can be affected by the particular debugging stage at which the rates are evaluated. Based on our experiment, we conjecture that the accuracy of a reliability prediction is affected by the fault recovery order as well as by fault interaction.

Keywords: Software Reliability Models, Prediction, Empirical Studies, Ultrareliable Software

*This research was partially supported by NASA grant NAG-1-750.

Introduction

Ultrareliable software is required for life critical applications such as the control of nuclear power or flight control. Efforts to meet and certify this level of reliability will require increased testing as well as more accurate data collections and computations. If software reliability models are to be useful in this quest we must refine and improve these models significantly. These requirements motivated us to investigate sources of inaccuracy in existing models with the hope of improving their performance and/or of discovering better models.

It has been observed that predicting the reliability of a program has proved to be an unexpectedly challenging task for over two decades [Bas93]. The first published descriptions of parametric models used to describe future software performance appeared in the early 1970's [JeM72, WiS72]. While many additional software reliability models have since been proposed, no one model has emerged as universally applicable; nor is it clear how to choose in advance a model for a given project.

Typically software reliability models use a sequence of interfailure times from the debugging process to predict reliability or the related quantities of failure rate and mean time to next failure. However the sequence of interfailure times is derived from only one of many possible repair orders. If one assumes data from n failures are being used, then there are n! possible orders in which those faults could have been individually identified and repaired. Also uncertainty about the order of fault recovery is compounded in that a sample of size one is used to represent the interfailure time of the software for each stage of the fault removal process. We investigate multiple orders for the fault removal process and isolate the effects of varying the recovery order by using an average of multiple observations to represent each interfailure time.

It has been conjectured in the literature that one failure may prevent access to or hide certain others. The manifestation of this has been referred to as the fault interaction phenomenon [Dun86]. Other researchers claim that in practice, although such interactions may occasionally occur during unit testing, they are much less common during system testing or in the operational phase [MIO87]. Our data displays some manifestations of fault interactions.

We believe that fault recovery order may affect the accuracy of predictions made by software reliability models and that the effects of bug interactions are subsumed by the recovery order problem, since for a particular recovery order the context in which each fault contributes to the failure rate is fixed. The remainder of this paper will focus on the use of the debugging graph to analyze the effect of recovery order on reliability predictions and is organized as follows. Preliminary considerations are in sections on terminology and models, on the debugging graph and on a description of the experiment. Data are described and presented in sections on fault sizes and path anomalies, and on path selections. There is a section analyzing the performance of the models followed by one on conclusions and issues.

Terminology and models

Software reliability (R) is the probability of a software product operating for a given period of time in a particular environment without exhibiting any failures. In many instances, particularly in the highly cyclic applications with periodic deadlines such as flight control systems, the number of input cases is proportional to the execution time. We will assume this to be true for the remainder of the paper. This will allow us to use the average time of computation for an input case as the given time period, and R becomes the probability of success per input. The failure rate (F = 1 - R) expresses the probability that a software product will exhibit a failure during a given time period in its specified environment. A third important quantity is the mean time to failure (MTTF = 1/F) which indicates the number of expected input cases prior to the next failure. We use the term fault recovery to mean the identification of faults and the implementation of suitable code repairs to remove them from the program.

The models examined in this experiment are Jelinski-Moranda [JeM72]; Geometric De-Eutrophication [Mor75]; Basic Musa [MIO87]; and Logarithmic Poisson [MuO84]. The Jelinski-Moranda model assumes that all faults contribute equally to the unreliability of the program, so that the plot of failure rate versus time is a step function in which each step essentially represents one "error's worth" of hazard. In an attempt to describe testing in which an accumulated group of faults is corrected simultaneously or the hazard contributions of faults are not equal, the Geometric De-Eutrophication model assumes a plot of failure rate versus time in which the step size decreases in a geometric sequence with each subsequent fault removal. The Basic Musa and Logarithmic Poisson models are continuous analogues to the Jelinski-Moranda and Geometric De-Eutrophication, respectively.

The Debugging Graph

Suppose a program contains n known faults labeled 1..n respectively. There are n! possible orders in which the n faults could have been individually located and repaired. The debugging graph is useful for representing and studying these n! orders. [WiS87]. The rows, or levels, of the debugging graph are labeled from 0 to n, with row i representing stage i of the debugging process where i of the n bugs have been repaired. The term variant references any version of the original program with some subset of the known repairs installed. Each graph node represents a variant and is labeled with P subscripted by the subset of {1,2,...,n} corresponding to the faults repaired in that variant.

There is a single node, labeled P, at level 0; it represents the variant with no repairs installed. Likewise, there is a single node at level n, labeled $P sub {1,....,n}$; it represents the software with all n known repairs installed. At level 1, there are n nodes, labeled $P sub 1$ through $P sub n$, each representing a variant of the original program P with only one of the n known faults removed. In general, at level m, $0 <= m <= n$, there are n! / (m! (n-m)!) nodes and a total of $2 sup n$ nodes in the debugging graph.

An edge in the graph represents a repair for one fault and connects one node to another at the next level, where the subscripts of the adjacent nodes differ in exactly one element. In general, at level m, $0 <= m <= n$, each node has (n-m) edges joining it to nodes at level (m+1). This results in a total of $2 sup {n+1}$ edges in the debugging graph. A debugging session removes all known bugs and is represented by a path in the debugging graph from P to $P sub {1...n}$ that follows edges through exactly one node at each of the levels 0 through n.

In order to obtain empirical reliability estimates each variant is subjected to a large set of inputs, generated randomly according to the prescribed usage distribution, and the number of inputs producing expected outputs is determined by using an oracle (error detector). This enables each variant's reliability to be estimated using the calculation: R = (number of "expected" outputs) / (total number of inputs). Thus each node in the debugging graph subsequently can be labeled with the empirically determined reliability of its corresponding variant.

Experiment Description

For this preliminary experiment we required non-trivial code developed under controlled scientific conditions for which a documented debugging history existed. To this end we chose a version of the Launch Interceptor Code (LIC) which was prepared by RTI [Dun86] as part of a previous NASA experiment. This code consisted of about 500 lines of FORTRAN written by a professional programmer to specifications which included the input distribution; it came with the required debugging history as well as a gold version (presumably correct) which could be used to develop an oracle. This version had twelve known bugs; however, we found that two of them were artifacts of the previous test environment and did not violate the specifications. The repairs for those two bugs were installed prior to collecting data. We then constructed a debugging graph for the remaining bugs (n = 10) by using 1 million input cases to estimate R for each of the 1024 nodes as described below.

The software was ported from the NASA Langley Research Center's AIRLAB facility in Hampton, Virginia to the CS Department Sun network at ODU. A testing environment, LICCtrl (see Figure 1), was tailored for our experiment to generate a large, random (but repeatable) input stream, and to run two separate subroutines: the gold version; and a selected LIC test variant containing some combination of known bugs and repairs. On each iteration, both subroutines were exercised using the same input values and the outputs were compared for equality. Output agreement corresponds to a successful or "expected" outcome; while output disagreement and/or abnormal termination of the test variant denotes a failure. Successful runs were tallied and R was calculated for the variant.

```
read runtime parameters;
for( desired number of runs ) {
generate next set of input values;
        load common block;
        call gold subroutine;
        load common block;
        call test subroutine;
        compare gold and test results;
        tally & record;
}
output summary statistics;
exit(0);
```

Figure 1. Overall LICCtrl Program Logic

Fault Sizes and Path Anomalies

Since intuitively the debugging process is most likely to recover bugs in a "largest-to- smallest" order, we conjectured that recovering the faults in various "size" orders is appropriate. We applied a criterion used in earlier experiments which associates bug sizes with their observed failure rates [Cow91]. Table 1 illustrates the reliability changes realized by installing each of the ten known repairs in the initial program P (level 1) versus those changes created by removing only one of the known repairs from P1,...,10 (level 9). At level 1, observed failure behavior for a given fault-inferred by observing the effects of installing its repair-is subject to the influence of all other faults, both known and unknown, in the program. At level 9, observed failure behavior for a given fault-inferred by observing the effects of removing its repair-is not subject to the influence of other known faults, but is influenced by as yet undiscovered faults in the program.

According to Table 1, the only faults which appear to behave identically at both levels are 7 and 10, with 8 and 9 exhibiting not much difference. In fact we noted that half of the ten bugs considered could potentially change ranking depending on whether the size comparison were performed at level 1 or at level 9. So we used both as possible size rankings when constructing comparison paths. (Note: We defer for now special handling of cases in which the R values are equal, such as bugs 9 and 10 at level 9, and simply use the ordinal ranking to break ties). Table 1 illustrates, as did earlier

Table 1. Some Bug Sizes

Level 1	Repair	Level 9
Reliability Change		*Reliability Change*
51.3188	1	54.6220
1.4534	2	2.7724
0.1368	3	0.1472
0.4365	4	1.1626
0.6057	5	1.0076
0.0051	6	0.0286
0.0000	7	0.0000
0.0003	8	0.0004
0.0000	9	0.0001
0.0001	10	0.0001

experiments, that by looking at multiple debugging graph levels the observed change in reliability associated with installing a particular repair can vary with the presence or absence of the other faults in the program [WhH90].

The previous paragraph revealed that even when considering a repair which is known to be "correct," one cannot always expect it to have the same reliability growth effect, since that effect depends on when the repair is installed during the debugging process. Examination of various levels of the debugging graph and the attempts to "size" the software's known faults revealed that not only do individual faults fail with different rates [NaS82], but that the rate experimentally associated with a given fault is a function of the program's debugging state at the time assessment is made. We attribute this effect to interactions among faults and note that these interactions may result in unexpected changes in the failure behavior of the variants.

Similarly, one might expect that the proportion of inputs producing expected outputs would always increase during a debugging session as each subsequent fault is repaired by a known "correct" fix. By inspecting sample debugging paths, however, we realized that installing a given repair can result in positive reliability growth, "low-to-no" change or even negative growth depending on the stage at which the repair is done. There may even be a subset of the known repairs which, when installed, results in a higher reliability figure than installing all known repairs. In the past, such behavior has been attributed to partial or incorrectly done repairs. The anomalies in reliability growth on certain paths lead us to alternative conclusions. Not only is the rate experimentally associated with a given fault subject to the program's debugging state, but the installation of a correct repair may result in a negative reliability growth. We believe these phenomena are physical manifestations of interaction effects that need to be taken into account during software reliability modeling.

Path Selections

For model comparison purposes, we wanted to choose a variety of debugging paths based on the graph construction effort described above. Debugging paths were constructed according to the fault recovery criteria enumerated in Table 2. The path pairs 1 & 3 and 2 & 4 were intentionally chosen to be intuitive (largest-to-smallest) and counter-intuitive (smallest-to- largest) recovery orders, respectively.

Paths 5 & 6 are the intuitive and counter-intuitive examples of hybrid paths. The intent of a hybrid is to size-rank the first half of the known faults with respect to their behavior in the otherwise unrepaired program (at level 1), and the remaining faults with respect to their behavior in the otherwise maximally repaired program (at level (n-1)). Such paths may more accurately reflect reality in the sense that early in the debugging process, bugs exist in the presence of many others; while later bugs are ranked in an isolated fashion to reflect the more purified nature of the software that exists at that time. Paths 7 through 10 were included to stress the predictive models by making the reliability improvements oscillate between relatively large and small steps. Paths 11 & 12 recover faults in mixed size orders, while Path 13 represents fault repair in the original recovery order.

Inspection of the empirically calculated R value sequences associated with each of these paths reveals that they collectively illustrate a variety of anomalies (e.g., flat or negative reliability growth sequences), intentionally "expected" and "unexpected" repair orders (e.g., largest-to- smallest versus smallest-to-largest), and predominately high as well as predominately low R values.

Table 2. Description of Debugging Paths

Path	*Construction Method*
1	largest-to-smallest order w.r.t. level 1 size rankings
2	smallest-to-largest order w.r.t. level 1 size rankings
3	largest-to-smallest order w.r.t. level 9 size rankings
4	smallest-to-largest order w.r.t. level 9 size rankings
5	five largest repairs in non-increasing size order w.r.t. level 1 size rankings, followed by the remaining five repairs in non-increasing size order w.r.t. level 9 size rankings
6	five smallest repairs in non-decreasing size order w.r.t. level 1 size rankings, followed by the remaining five repairs in non-decreasing size order w.r.t. level 9 size rankings
7	alternate the largest remaining repair followed by the smallest remaining repair w.r.t. level 1 size rankings
8	alternate the smallest remaining repair followed by the largest remaining repair w.r.t. level 1 size rankings
9	alternate the largest remaining repair followed by the smallest remaining repair w.r.t. level 9 size rankings
10	alternate the smallest remaining repair followed by the largest remaining repair w.r.t level 9 size rankings
11	medium, small and large repairs in mixed order w.r.t. level 1 size rankings
12	medium, small and large repairs in mixed order w.r.t. level 9 size rankings
13	original order (1,2,3 etc.)

The Jelinski-Moranda [JeM72], Geometric De-Eutrophication [Mor75], Basic Musa [MIO87], and Logarithmic Poisson [MuO84] models were implemented in the C programming language and executed on Sun SparcStations. We used the relationship MTTF = 1 / (1 - R) to generate input sequences for each chosen path. These sequences were used as inputs for each model, and the failure time s predicted by the models were compared against the known, empirically observed failure data. Along each chosen debugging path, (i+1) experimentally generated consecutive MTTFs were used as inputs to a model to predict the (i+1)st failure time for each i from 1 to 10. (Note: The 0th input value used was derived from the R value associated with the variant P.)

As a measure of a model's predictive accuracy, the predicted failure time was in each case compared with the corresponding empirically determined MTTF calculated from the R value determined from the data for the appropriate variant. The comparison was normalized by taking the ratio of estimated MTTF as predicted by a model to the empirical MTTF. Ratios close in value to 1 are interpreted as indicating accurate predictions, while values greater than 1 indicate optimistic predictions and those between 0 and 1 indicate pessimistic predictions. For example, a predicted MTTF that is an order of magnitude greater than the actual MTTF produces a ratio value near 10. This implies that the software would probably fail ten times sooner than one would have expected based on the estimated MTTF value; hence the model's prediction is optimistic.

Performance Analysis

The prediction ratios for the four models are shown in Tables 3 through 6. The tables should be read in a row-wise fashion, where each row represents predictions along the debugging path cited in its left-most column. Six decimal places are carried to reflect the precision of the MTTF values input to the models. Nine predictions are presented for each path, they are labeled in the tables as "MTTF Prediction Stages." These range from the prediction of mean time to second failure to the prediction of the mean time to tenth failure. Although we can make a prediction of mean time to eleventh failure, without additional debugging data we lack the empirical R value needed to calculate a comparison ratio in this case. We marked exceptional cases with asterisks; appropriate interpretations are indicated with each table.

The underlying assumption of the Jelinski-Moranda algorithm-that "all bugs are created equal"-probably makes it inappropriate for realistic applications. It is interesting that the model's predictions were initially quite good for the "mixed" recovery order paths (11 and 12). Although it otherwise generally failed at prediction, the algorithm performed consistently over the "intuitive" paths, assessing a finite number of bugs after the first few iterations; whereas "counter-intuitive" paths proved to be more challenging, probably due to "low-to-no" growth characteristics.

For several "counter-intuitive" paths (paths 2, 4 and 6), the Geometric De- Eutrophication algorithm provided the most accurate overall predictions possibly since the effect of the largest fault was postponed in those particular cases. It also gave good predictions for "mixed" recovery order paths (11 and 12) until the last two predictive stages where the performance again degenerated after the data representing the largest fault was introduced. Along the remaining paths, the model appeared to try and correct its predictions at some point; but the predictions tended to grow either increasingly optimistic or pessimistic thereafter.

The behavior of the Basic Musa model, as compared to that of its discrete counterpart Jelinski-Moranda, can be viewed as highly inconsistent from path to path as well as along any given path. Although it made predictions, all were either very optimistic or pessimistic, and it often incorrectly predicted perfect software after a large relative reliability improvement step. Interestingly, however, the influence of the very large repair (i.e., the fix for bug 1) appeared to be mitigated if it were inserted early.

The primary challenge in using the Logarithmic Poisson model was determining parametric values which "fit" the functions, given the input data precision and host computer accuracy; this problem was particularly evident when "small-to-large" or "mixed" repair orders were p resented as inputs. However this model performed extremely well on the "intuitive" paths (1,3 and

5) and the original order path (13), with possibly a very slight preference for the intuitive paths. It was the only model to perform well on the intuitive path. We note also the closeness of the predictions of this model and the Basic Musa model on portions of some paths (e.g., paths 11 and 12).

Table 3. Jelinski-Moranda Prediction Ratios

** no solution; N is infinite or no R growth present*

*** no solution; N is finite*

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	*	**	**	**	**	**	**	**	**
2	*	*	*	*	*	*	.984969	.966156	1.61788e-5
3	*	**	**	**	**	**	**	**	**
4	*	*	*	*	*	*	.986483	.967225	1.61919e-5
5	*	**	**	**	**	**	**	**	**
6	*	*	*	*	*	*	.986483	.967225	1.61919e-5
7	*	*	**	**	**	4.30056	**	**	**
8	*	**	**	**	**	**	**	**	**
9	*	*	**	**	**	**	**	**	**
10	*	**	**	**	**	**	**	**	**
11	1.00231	.993849	1.00127	.992366	1.00038	.977324	.992851	1.83469e-5	**
12	1.00231	1.00162	.990453	.990931	.999741	.977105	.992865	1.65146e-5	**
13	*	**	**	**	**	**	**	**	**

Table 4. Geometric De-Eutrophication Prediction Ratios

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	5.254125	4.094815	0.653375	0.525571	0.131187	1.722893	6.503249	26.333447	64.380750
2	1.000000	0.999998	0.999995	0.999912	0.997586	0.991224	0.984954	0.966084	0.000016
3	5.254125	3.636669	0.728972	0.544683	0.133134	1.731698	6.522468	23.720902	64.505362
4	1.000000	0.999998	0.999995	0.999912	0.997586	0.988216	0.986459	0.967144	0.000016
5	5.254125	4.094815	0.653375	0.525571	0.131187	1.722893	6.503249	23.700102	64.607380
6	1.000000	0.999998	0.999995	0.999912	0.997586	0.988216	0.986459	0.967144	0.000016
7	11.434335	2.787241	2.999700	1.656010	2.123555	0.315943	1.102452	0.316568	0.031772
8	0.087456	2.587646	1.845143	3.083805	2.130480	2.782936	0.423873	1.044780	0.011590
9	11.434335	2.787241	2.999700	1.470728	2.045788	0.357346	1.151743	0.326621	0.032295
10	0.087456	2.587646	1.845143	3.083805	1.892095	2.628615	0.469600	0.217302	0.030771
11	1.002302	0.993848	1.001238	0.992335	1.000283	0.977219	0.992475	0.000018	0.622673
12	1.002302	1.001619	0.990452	0.990893	0.999628	0.976982	0.992461	0.000017	0.699562
13	5.254125	6.713279	2.491269	0.087452	0.087452	2.245986	6.142692	22.041007	53.985926

Table 5. Basic Musa Prediction Ratios

** indicates software predicted to be perfect*

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	0.229752	0.191542	0.627602	5.12068e+10	*	204.19	3.42157	0.480884	0.174262
2	0.5	0.333333	0.249999	0.199983	0.166269	0.141748	0.123628	0.108139	1.64873e-6
3	0.229752	0.170157	3.04723	3.46102e+10	*	203.974	3.42072	0.564199	0.221082
4	0.5	0.333333	0.249999	0.199983	0.166269	0.141318	0.124005	0.108139	1.64873e-6
5	0.229752	0.191542	0.627602	1.5362e+07	*	204.19	3.42157	0.56425	0.221097
6	0.5	0.333333	0.249999	0.199983	0.166269	0.141318	0.124005	0.108139	1.64873e-6
7	0.5	0.153168	0.26595	0.114748	0.229378	0.0191406	4.35596	567340	*
8	0.0437279	0.700702	0.114876	0.394062	0.0957884	0.307529	0.31857	18.2149	*
9	0.5	0.153168	0.26595	0.10191	0.286481	0.0215521	3.91331	385886	*
10	0.0437279	0.700702	0.114876	0.394062	0.0850967	0.387969	1.54892	3.31808e+12	*
11	0.499955	0.330746	0.249999	0.197805	0.166666	0.139036	0.125	2.03437e-06	598.426
12	0.499955	0.333332	0.247306	0.198407	0.166666	0.139036	0.125	1.83192e-06	219.068
13	0.229752	0.31376	0.119647	1.67047e+10	*	16.3482	3.65748	0.671158	0.275259

Table 6. Logarithmic Poisson Prediction Ratios

** indicates no solution for desired precision*

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	0.836174	0.19322	0.253012	0.994774	1.17794	1.52371	1.35359	1.1899	1.01764
2	0.500002	0.333334	0.25	*	0.16627	*	0.123629	0.10814	*
3	0.836174	0.171919	0.384522	0.991078	1.1759	1.5226	1.35247	1.1887	1.02948
4	0.500002	0.333334	0.25	*	0.16627	*	0.124005	0.10814 *	
5	0.836174	0.193228	0.253012	0.994774	1.17794	1.52371	1.35359	1.1899	1.03081
6	0.500002	0.333334	0.25	*	0.16627	*	0.124005	0.10814	*
7	0.954732	*	0.275298	*	*	*	0.553978	*	0.645629
8	0.0437315	0.899645	0.418087	0.514248	0.0967482	*	0.126748	0.591096	0.52569
9	0.954732	*	0.275298	*	0.276793	*	0.550244	*	0.643364
10	0.0437315	0.899645	0.418087	0.514248	0.0860996	0.385635	0.192425	0.587578	0.750928
11	*	0.330746	0.249999	0.197806	0.166667	0.139037	0.125	2.48459e-06	*
12	*	0.333332	*	0.198408	0.166667	0.139037	0.125	*	*
13	0.836174	0.315014	*	0.231535	1.19901	1.53566	1.3109	1.18671	1.03164

The data derived from comparing the four reliability models clearly show that along a given debugging path, the predictive performance of these models can vary greatly (e.g., Path 1 in Tables 3 through 6). Additionally, just because a model appears to "fit" a given path's data well in terms of its predictive performance, there is no guarantee that the model would still appear as appropriate had those faults been recovered in a different order (e.g., contrast paths 4 & 7 in Table 4). In other words, if the experimenter evaluates models based on a single realization of the debugging process, with the faults recovered and corrected in a single order, he or she might reject a model that could perform quite well using data from an alternative recovery order.

Conclusions and Issues

We have presented a methodology involving the debugging graph whereby the performance of software reliability models can be analyzed under laboratory conditions. This procedure allows one to use the average of large sample sets to replace the single point samples normally used as input for these models and thus to scrutinize their performances with some of the randomness removed from the input data.

In this preliminary experiment we observed that recovery order is an important factor in the potential performance of these models. Further, based on this work it appears that if one could pick the model, control the recovery order and use the average of large samples for the interfailure times, then one could expect more accurate predictions. Further experiments are needed involving new specimens and problems to support this conjecture. Also it is a challenge to implement improvements in the software prediction process based on these ideas.

We observed multiple manifestations of bug interactions in this data. Further exploration of the debugging graph database should be conducted to study side-effects of bug interactions; for example, a debugging path along which the outputs from selected inputs oscillate between agreeing and disagreeing with the oracle may provide new clues as to the nature of bug interaction.

References

[ACL86] Abdalla A. Abdel-Ghaly, P. Y. Chan and Bev Littlewood, "Evaluation of Competing Software Reliability Predictions," IEEE Transactions on Software Engineering, vol. SE-12, no. 9, September 1986, pp. 950-967.

[Bas93] Farokh Bastani, "Forward: Software Reliability," IEEE Transactions on Software Engineering, vol. 19, no. 11, November 1993, pp. 1013-1014.

[Cow91] Christopher Cowles, "Measuring Software Reliability Models," Master's Degree Project Report (unpublished), Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1991.

[Dun86] Janet R. Dunham, "Experiments in Software Reliability: Life-Critical Applications," IEEE Transactions on Software Engineering, vol. 12, no. 1, January 1986, pp. 110-123.

[JeM72] Z. Jelinski and P. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation, Walter Freiberger, (ed.). New York: Academic Press, 1972, pp. 465-483.

[Mor75] P. B. Moranda, "Prediction of Software Reliability During Debugging," Proceedings of the Annual Reliability and Maintainability Symposium, 1975, pp. 327-332.

[MIO87] John D. Musa, Anthony Iannino and Kazuhira Okumoto, Software Reliability: Measurement, Prediction, Application. New York: McGraw-Hill Book Company, 1987.

[MuO84] J. D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," 7th IEEE International Conference on Software Engineering, 1984, pp. 230-238.

[NaS82] Phyllis M. Nagel and James A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling," CR-165836, NASA Langley Research Center, Hampton, Virginia, February 1982.

[WhH90] Richard L. White and Christine F. Harbison, "The Error Graph: Research in Software Reliability," Master's Degree Project Report (unpublished), Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1990.

[WiS72] K. G. Wilkinson and M. L. Shooman, "Probalistic Models for Software Reliability Prediction," Statistical Computer Performance Evaluation, Walter Freiberger, (ed.). New York: Academic Press, 1972, pp. 485-502.

[WiS87] Larry Wilson and Wenhui Shen, "Software Reliability Perspectives," TR-87-035, Old Dominion University, Norfolk, Virginia, 1987.